Lecture 8 Red Black Trees, Hashing
Scribes: Vishnu Sundaresan (2015), Virginia **Date:** April 24, 2016

1 Red-Black Trees

Today, we'll finish off our coverage of red-black trees. Recall from last time that red-black trees maintain the following invariants.

(1) Every node is red or black

(2) Root and NILs are black

(3) Both children of a red node are black

(4) For any node x, all $x \to NIL$ paths have the same number of black nodes

We showed last time that, if we are able to maintain these properties, then the tree will remain balanced, and all searching through the tree will take $O(\log n)$ time. Here is some more intuition on why the tree is roughly balanced.

Intuition: By Invariant (4) above, all $r \to NIL$ paths have $b(r)$ black nodes (excluding the root). Therefore, all these paths have length $\geq b(r)$. However, they also have length $\leq 2 \cdot b(r)$: by Invariant (3), the number of red nodes is limited to half of the path, since every red node must be followed by a red node, and hence the number of black nodes is at least half of the length of the path. Hence the lengths of all paths from r to a NIL are within a factor of 2 of each other and the tree must be reasonably balanced.

In what follows, we'll describe how we maintain these invariants through insertion in $O(\log n)$ time. Our coverage here is detailed, but not comprehensive, and meant as a case study. For complete coverage, please refer to Ch. 13 of CLRS.

1.1 Insertion in a Red Black Tree

The process for inserting a new node is initially similar to that of insertion into any BST.

Algorithm 1: Insert (i)

```
$p \leftarrow BST - Search(i)$;
$x \leftarrow$ new node with key $i$, child of $p$;
color $x$ red;
recolor if necessary;
```

Note that when x is inserted as a red node: Invariant (1) is satisfied, as we colored the new node red; Invariant (2) is satisfied, as we did not touch the root; and Invariant (4) is satisfied, as we did not change the number of black nodes in the tree. Thus, the only invariant we have to worry about is Invariant (3). If p is black, then this invariant is also maintained, so we can return. Otherwise, we have a red child of a red node, so we must recolor the tree.

We'll consider the case when x is a left child of p and p is a left child of its parent, p'. This case is representative, showing most of the machinery that we'll need to insert an arbitrary element into an arbitrary red-black tree.

[Besides the case where x is a left child of p and p is a left child of p', we see that there are 3 other possible cases (whether or not x is the left or right child, and whether or not p is the left or right child), but we will not cover those in this lecture.]

The key to insertion is following the so-called “uncle” of x, which we call u – that is, the other child of p' besides p. We will have two cases, one when the uncle is red, one when it is black.

Algorithm 2: Recolor (x)   `// x is a left child, p is a left child`

```
$p \leftarrow$ parent of $x$;
if $p$ is black then
    return;
$p' \leftarrow$ parent of $p$;
$u \leftarrow$ right child of $p'$;
if $u$ is red then
    color $p$ and $u$ black;
    color $p'$ red;
    recolor ($p'$);
else if $u$ is black then
    color $p$ black;
    color $p'$ red;
    right-rotate ($p'$);
```

Note that when u is red, we switch the colors of p and u to black and the color of p' to red. If the parent of p' is black, then Invariant (3) is maintained; additionally, because we switched the colors of two nodes on each of these paths (one red$\rightarrow$black and one black$\rightarrow$red), the number of black nodes on each path is unchanged, so Invariant (4) remains unchanged. If the parent of p' is red, then we must recolor starting at p' to resolve Invariant (3).

When u is black, we recolor p to black and p' to red and then right-rotate around p'. We can see that this also maintains the same number of black nodes on each path, and satisfies Invariant (3) below p because the original tree was a red-black tree.

This recursion will bottom out when we hit the root, with a constant number of relabelings and rotations at each level, so it will be an $O(\log n)$ operation overall since we showed that the height of the tree is $O(\log n)$. Thus, we can update our red-black trees in $O(\log n)$ time upon insertion. The other operations are similar, and also give the guarantee of worst-case performance of $O(\log n)$ insertion, deletion, and look-up.

As we have seen, BSTs are very nice – they allow us to maintain a set and report membership, insert, and delete in $O(\log n)$ time. In addition to these basic underlying operations, we can also support a slew of other types of queries efficiently. Because the elements are stored maintaining the binary search tree property, we can search for the next largest element or the elements on a range very efficiently. But what if we don't care about these properties? What if we only need to support membership queries? Can we improve our performance of $O(\log n)$ time to nearly constant time? This question motivates our discussion of hash tables.

2 Hash Tables

Let S be a set of “records”, with a key and data associated with each key. We want to support the following operations:

- search(k): return $x \in S$ with $key(x) = k$ or NIL if $x \notin S$
- insert(x): $S \leftarrow S \cup \{x\}$
- delete(x): $S \leftarrow S - \{x\}$

Binary search trees solve all of these operations in $O(\log n)$ time, but we'd like to do even better.

2.1 Direct Access Tables

Suppose the keys are distinct and are drawn from the set $U = \{0, \ldots, n-1\}$. Then, we can keep an array $T\{0, \ldots, n-1\}$ such that $T(k) = x$ if the key of x is k. This structure is called a direct access table, and has $O(1)$ insert, delete, and search.

Initially this solution seems great – all of the functionality we desire, all with constant runtime! But, this solution also clearly has its limitations. Note that the range of keys is typically $\leq 2^w$ where w is the computer word size (32 or 64 bits on a modern computer). Storing a direct access table for all of these keys is completely infeasible, as the size required to store this access table is enormous: 2^{64} is 18 quintillion. Another more subtle limitation is the following. Suppose we are trying to have keys roughly corresponding to words in the English language. A direct access table is also not very ideal in this situation. As the data has a very regular structure, if we created a cell for each possible key (say of all strings on ≤ 10 characters), the resulting table would have the majority of its entries empty. Is there a way we can trade off these limitations?

2.2 Hash Functions

This is where the concept of hashing comes into play. Instead of requiring the array to be large enough to cover all possible keys, we'll set the size to be smaller, and use "hashing" to allow us to get almost comparable performance.

For every key in our universe U, we will apply some hash function h from $\{0, \ldots, |U-1|\}$ to $\{0, \ldots, m-1\}$, where m will be the size of our table. When a given key x comes in, we will first "hash" x, computing $h(x)$ and place x into the corresponding array slot, $T[h(k)]$.

This solution introduces an inherent problem: by the pigeonhole principle, when $|U| > m$, there will be collisions, when two keys $k_1 \neq k_2$ are mapped by the hash function to the same value, $h(k_1) = h(k_2)$. There are two ways to handle collisions:

1. chaining (will be discussed today)
2. open addressing (addressed next time)

2.3 Chaining

In order to deal with the fact that we may need to store multiple records in each hash table slot, instead of storing a single record in each array entry, we will now store a list of records. That is, $T[i]$ will contain NIL if there are no records in S with key hashed to i, and will otherwise contain a list of all records whose keys are hashed to i. This way of resolving collisions is called *chaining*.

2.3.1 Worst Case Analysis

To insert an element, we simply insert the new record at the end of the list of records in the slot that it is mapped to. Thus, the insertion time is still constant.

However, to search for or delete an item with key k, one needs to first compute $h(k)$, access $T[h(k)]$ and then search the list stored there from the beginning until a record with key k is found. This process could require you to look through the entire list of records at the given hash value slot, which could, in principle, be all of the records stored. We can imagine an "adversary" who knows the hash function we are using, picking out inputs which will all map to the same value. Thus, the resulting worst-case running time for search is $\Theta(n)$. Implementing deletion requires searching for the element to delete, so again, this operation could take $\Theta(n)$ time.

2.3.2 Average Case Analysis

To combat this worst-case performance, we will make the following assumption about our hash functions. We say that the hash function h satisfies the property of *simple uniform hashing*, if each key $k \in S$ is equally likely to be hashed to any slot in the range of h, $\{0, \ldots, m-1\}$, independent of where the other keys are hashed.

To analyze the performance of a hash table using a simple uniform hash function, we introduce the following definition.

Definition 2.1. *The load factor α of a hash table with n keys and m slots is $\alpha = (n/m)$.*

Note that α is the average length of a chain in the hash table.

Claim 1. *The expected time of an unsuccessful search is $O(1+\alpha)$*

Notice that $O(1+\alpha) \leq O(1)$ if and only if $n \leq O(m)$ since $\alpha = n/m$. Thus, the expected search time is $O(1)$ if and only if the choice m for the number of slots is $\Omega(n)$. Thus picking the size of the hash table is crucial- if one picks m to be too small, the search time won't be constant anymore. If m is picked too big, however, then the search time will be fast in expectation, however much of the table will be empty.

Proof of Claim 1. Since one needs to at the very least compute $h(k)$, one needs to spend at least $O(1)$ time in any search call. To ascertain that no record with key k is in the hash table, one needs to read all records in the chain stored in slot $h(k)$. The length of this chain is $O(\alpha)$ in expectation. Thus the expected search time is $O(1+\alpha)$. $\square$